# 加餐 | 部分课后思考题答案合集

2020-02-12 吴咏炜

现代C++实战30讲 进入课程 >

**讲述：吴咏炜**
时长 11:38　大小 10.66M

你好，我是吴咏炜。这一讲我为你整理了本专栏部分课后思考题的答案，给你作为参考。

## 第 2 讲

**你觉得智能指针应该满足什么样的线程安全性？**

答：（不是真正的回答，只是描述一下标准中的智能指针的线程安全性。）

1. 多个不同线程同时访问不同的智能指针（不管是否指向同一个对象）是安全的。
2. 多个不同线程同时读取同一个智能指针是安全的。
3. 多个不同线程在同一个智能指针上执行原子操作（`atomic_load` 等）是安全的。

4. 多个不同线程根据同一个智能指针创建新的智能指针（增加引用计数）是安全的。
5. 只会有一个线程最后会（在引用计数表示已经无引用时）调用删除函数去销毁存储的对象。

其他操作潜在是不安全的，特别是在不同的线程对同一个智能指针执行 `reset` 等修改操作。

## 第 3 讲

**为什么 `smart_ptr::operator=` 对左值和右值都有效，而且不需要对等号两边是否引用同一对象进行判断？**

答：我们使用值类型而非引用类型作为形参，这样实参永远会被移动（右值的情况）或复制（左值的情况），不可能和 `*this` 引用同一个对象。

## 第 4 讲

**为什么 `stack`（或 `queue`）的 `pop` 函数返回类型为 `void`，而不是直接返回容器的 `top`（或 `front`）成员？**

答：这是 C++98 里、还没有移动语义时的设计。如果 `pop` 返回元素，而元素拷贝时发生异常的话，那这个元素就丢失了。因而容器设计成有分离的 `top`（或 `front`）和 `pop` 成员函数，分别执行访问和弹出的操作。

有一种可能的设计是把接口改成 `void pop(T&)`，这增加了 `T` 必须支持默认构造和赋值的要求，在单线程为主的年代没有明显的好处，反而带来了对 `T` 的额外要求。

## 第 5 讲

**为什么大部分容器都提供了 `begin`、`end` 等方法？**

答：容器提供了 `begin` 和 `end` 方法，就意味着是可以迭代（遍历）的。大部分容器都可以从头到尾遍历，因而也就需要提供这两个方法。

**为什么容器没有继承一个公用的基类？**

答：C++ 不是面向对象的语言，尤其在标准容器的设计上主要使用值语义，使用公共基类完全没有用处。

## 第 7 讲

**目前这个输入行迭代器的行为，在什么情况下可能导致意料之外的后果？**

答：

复制代码

```
#include <fstream>
#include <iostream>
#include "istream_line_reader.h"

using namespace std;

int main()
{
  ifstream ifs{"test.cpp"};
  istream_line_reader reader{ifs};
  auto begin = reader.begin();
  for (auto it = reader.begin();
    it != reader.end(); ++it) {
    cout << *it << '\n';
  }
}
```

以上代码，因为 begin 多调用了一次，输出就少了一行……

**请尝试一下改进这个输入行迭代器，看看能不能消除这种意外。如果可以，该怎么做？如果不可以，为什么？**

答：很困难。比如，文件如果为空的话，从迭代器的行为角度，`begin()` 应该等于 `end()`——不预先读取一次的话，就无法获知这个结果。这样的改造总体看起来很不值，因此一般都不会选择这样做。

## 第 10 讲

**这讲里我们没有深入讨论赋值；请你思考一下，如果例子里改成赋值，会有什么样的变化？**

答：返回对象部分的讨论没有变化。对象的移动赋值操作应当实现成无异常，以确保数据不会丢失。

返回值优化在赋值情况下会失效。更一般的情况下，除非需要持续更新某个变量，比如在 `vector` 尾部追加数据，尽量对变量进行一次性赋值、不后续修改。这样的代码更容易推理，更不容易在后续修改中出错，也更能让编译器做（返回值）优化。

## 第 11 讲

**为什么说 UTF-32 处理会比较简单?**

答：UTF-32 下，一个字符就是一个基本的处理单位，一般不会出现一个字符跨多个处理单位的情况（UTF-8 和 UTF-16 下会发生）。

**你知道什么情况下 UTF-32 也并不那么简单吗?**

答：Unicode 下有所谓的修饰字符，用来修饰前一个字符。按 Unicode 的处理规则，这些字符应该和基本字符一起处理（如断行之类）。所以 UTF-32 下也不可以在任意单位处粗暴断开处理。

**哪种 UTF 编码方式空间存储效率比较高?**

答：视存储的内容而定。

比如，如果内容以 ASCII 为主（如源代码），那 UTF-8 效率最高。如果内容以一般的中文文本为主，那 UTF-16 效率最高。

## 第 12 讲

**为什么并非所有的语言都支持这些不同的多态方式?**

答：排除设计缺陷的情况，语言支持哪些多态方式，基本上取决于语言本身在类型方面的特性。

以 Python 为例，它是动态类型的语言。所以它不会有真正的静态多态。但和静态类型的面向对象语言（如 Java）不同，它的运行期多态不需要继承。没有参数化多态初看是个缺陷，但由于 Python 的动态参数系统允许默认参数和可变参数，并没有什么参数化多态能做得到而 Python 做不到的事。

## 第 17 讲

**想一想，你如何可以实现一个惰性的过滤器?**

答:

复制代码

```
#include <iterator>

using namespace std;

template <typename I, typename F>
class filter_view {
public:
  class iterator {
  public:
    typedef ptrdiff_t
      difference_type;
    typedef
      typename iterator_traits<
        I>::value_type value_type;
    typedef
      typename iterator_traits<
        I>::pointer pointer;
    typedef
      typename iterator_traits<
        I>::reference reference;
    typedef forward_iterator_tag
      iterator_category;

    iterator(I current, I end, F cond)
      : current_(current)
      , end_(end)
      , cond_(cond)
    {
      if (current_ != end_ &&
          !cond_(*current_)) {
        ++*this;
      }
    }
    iterator& operator++()
```

```
35      {
36        while (current_ != end_) {
37          ++current_;
38          if (cond_(*current_)) {
39            break;
40          }
41        }
42        return *this;
43      }
44      iterator operator++(int)
45      {
46        auto temp = *this;
47        +++this;
48        return temp;
49      }
50      reference operator*() const
51      {
52        return *current_;
53      }
54      pointer operator->() const
55      {
56        return &*current_;
57      }
58
59      bool operator==(const iterator& rhs)
60      {
61        return current_ == rhs.current_;
62      }
63      bool operator!=(const iterator& rhs)
64      {
65        return !operator==(rhs);
66      }
67
68    private:
69      I current_;
70      I end_;
71      F cond_;
72    };
73    filter_view(I begin, I end,
74                F cond)
75      : begin_(begin)
76      , end_(end)
77      , cond_(cond)
78    {}
79    iterator begin() const
80    {
81      return iterator(begin_, end_, cond_);
82    }
83    iterator end() const
84    {
85      return iterator(end_, end_, cond_);
86    }
```

```
87
88  private:
89    I begin_;
90    I end_;
91    F cond_;
92  };
```

## 第 18 讲

**我展示了 compose 带一个或更多参数的情况。你觉得 compose 不带任何参数该如何定义？它有意义吗？**

答：

复制代码

```
1  inline auto compose()
2  {
3      return [](auto&& x) -> decltype(auto)
4      {
5          return std::forward<decltype(x)>(x);
6      };
7  }
```

这个函数把参数原封不动地传回。它的意义相当于加法里的 0，乘法里的 1。

在普通的加法里，你可能不太需要 0；但在一个做加法的地方，如果别人想告诉你不要做任何操作，传给你一个 0 是最简单的做法。

**有没有可能不用 index_sequence 来初始化 bit_count？如果行，应该如何实现？**

答：似乎没有通用的办法，因为目前 constexpr 要求在构造时直接初始化对象的内容。

但是，到了 C++20，允许 constexpr 对象里存在平凡默认构造的成员之后，就可以使用下面的写法了：

复制代码

```
1  template <size_t N>
2  struct bit_count_t {
```

```
  constexpr bit_count_t()
  {
    for (auto i = 0U; i < N; ++i) {
      count[i] = count_bits(i);
    }
  }
  unsigned char count[N];
};

constexpr bit_count_t<256>
  bit_count;
```

当前已经发布的编译器中，我测下来只有 Clang 能（在 C++17 模式下）编译通过此代码。GCC 10 能在使用命令行选项 `-std=c++2a` 时编译通过此代码。

**作为一个挑战，你能自行实现出 `make_integer_sequence` 吗?**

答 1:

复制代码

```
template <class T, T... Ints>
struct integer_sequence {};

template <class T>
struct integer_sequence_ns {
  template <T N, T... Ints>
  struct integer_sequence_helper {
    using type =
      typename integer_sequence_helper<
        N - 1, N - 1,
        Ints...>::type;
  };

  template <T... Ints>
  struct integer_sequence_helper<
    0, Ints...> {
    using type =
      integer_sequence<T, Ints...>;
  };
};

template <class T, T N>
using make_integer_sequence =
  typename integer_sequence_ns<T>::
    template integer_sequence_helper<
      N>::type;
```

如果一开始写成 `template <class T, T N, T... Ints> struct integer_sequence_helper` 的话，就会遇到错误“non-type template argument specializes a template parameter with dependent type ‘T’ ”（非类型的模板实参特化了一个使用依赖类型的‘T’的模板形参）。这是目前的 C++ 标准所不允许的写法，改写成嵌套类形式可以绕过这个问题。

答 2：

复制代码

```
template <class T, T... Ints>
struct integer_sequence {};

template <class T, T N, T... Is>
auto make_integer_sequence_impl()
{
  if constexpr (N == 0) {
    return integer_sequence<
      T, Is...>();
  } else {
    return make_integer_sequence_impl<
      T, N - 1, N - 1, Is...>();
  }
}

template <class T, T N>
using make_integer_sequence =
  decltype(
    make_integer_sequence_impl<
      T, N>());
```

这又是一个 constexpr 能简化表达的例子。

## 第 19 讲

**并发编程中哪些情况下会发生死锁？**

答：多个线程里，如果没有或不能事先约定访问顺序，同时进行可阻塞的资源访问，访问顺序可以形成一个环，就会引发死锁。

可阻塞的资源访问可能包括（但不限于）：

- 互斥量上的 `lock` 调用
- 条件变量上的 `wait` 调用
- 对线程的 `join` 调用
- 对 `future` 的 `get` 调用

## 第 27 讲

**你觉得 C++ REST SDK 的接口好用吗？如果好用，原因是什么？如果不好用，你有什么样的改进意见？**

答：举几个可能的改进点。

C++ REST SDK 的 `uri::decode` 接口设计有不少问题：

- 最严重的，不能对 query string 的等号左边的部分进行 `decode`；只能先 `split_query` 再 `decode`，此时等号左边已经在 `map` 里，不能修改——要修改需要建一个新的 `map`。
- 目前的实现对“+”不能重新还原成空格。

换个说法，目前的接口能正确处理“/search?q=query%20string”这样的请求，但不能正确处理“/search?%71=query+string”这样的请求。

应当有一个 `split_query_and_decode` 接口，同时执行分割和解码。

另外，`json` 的接口也还是不够好用，最主要是没有使用初始化列表的构造。构造复杂的 JSON 结构有点啰嗦了。

`fstream::open_ostream` 缺省行为跟 `std::ofstream` 不一样应该是个 bug。应当要么修正接口（接口缺省参数里带上 `trunc`），要么修正实现（跟 `std::ofstream` 一样把 `out` 当成 `out|trunc`）。

**“概念”可以为开发具体带来哪些好处？反过来，负面的影响又可能会是什么？**

答：对于代码严谨、具有形式化思维的人，“概念”是个福音，它不仅可以大量消除 SFINAE 的使用，还能以较为精确和形式化的形式在代码里写出对类型的要求，使得代码变得清晰、易读。

但反过来说，“概念”比鸭子类型更严格。在代码加上概念约束后，相关代码很可能需要修改才能满足概念的要求，即使之前在实际使用中可能已经完全没有问题。从迭代器的角度，实际使用中最小功能集是构造、可复制、*、前置 ++、与 sentinel 类型对象的 !=（单一形式）。而为了满足迭代器概念，则要额外确保满足以下各点：

可默认初始化

在 iterator 类型和 sentinel 类型之间，需要定义完整的四个 == 和 != 运算符

定义迭代器的标准内部类型，如 `difference_type` 等

以上就是今天的全部内容了，希望能对你有所帮助！如果你有更多问题，还是请你在留言区中提出，我会一一解答。

课程学习计划

关注极客时间服务号
每日学习签到

月领 25+ 极客币

【点击】保存图片，打开【微信】扫码>>>

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇　30 | Coroutines：协作式的交叉调度执行

下一篇　结束语 | 终点已达，行程还要继续

## 精选留言 (2)

写留言

**晚风•和煦**
2020-02-13

老师，c语言可以通过哪些方式实现c++中的私有成员呢？谢谢老师😃😃😃

作者回复: 实现？不太明白你的意思了。私有只是编译时的访问控制，不是运行时的。而C完全没有编译时的访问控制的机制。

2

**幻境之桥**
2020-02-12

"如果内容以一般的中文文本为主，那 UTF-16 效率最高。"
这是为什么呢？中文不应该是 GBK更省空间吗？

展开

作者回复: 问题是“哪种 UTF 编码方式空间存储效率比较高”。GBK 不能支持很多其他语言，不在考虑范围内。

1